Warmup problem (L18)

- Find approximately the median temperature in nino34.csv by boolean masking instead of sorting
- That is, start with a guess, then in a loop, adjust the value until using it as a mask selects half the values

Fun with masks

CS114 (M8)

Extended example

- Let’s look for outlier temperatures in nino34.csv
- There are many definitions of “outlier”, but we’ll use “more than two standard deviations from the mean”

Step one: Read it in

- We need to read in the temperatures
- We also want a label for each temperature (the month during which the temperature occurred), so we'll read the year and month too
- We can't append these into an array, so we'll build lists

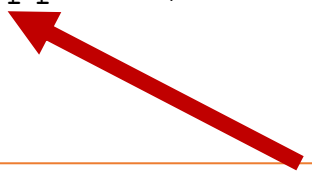
Step one: Read it in

```
monsLst = []
tempsLst = []
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    for row in nino:
        monsLst.append(f"{row['YR']}-{row['MON']}")
        tempsLst.append(float(row["TOTAL"]))
```

Be careful of strings in format strings. This is why strings can have either kind of quote!

Step one: Read it in

```
monsLst = []
tempsLst = []
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    for row in nino:
        monsLst.append(f"{row['YR']}-{row['MON']}")
        tempsLst.append(float(row["TOTAL"]))
```



The only thing that's associating a month with a temperature is that they're at the same index in each list. We could store dictionaries or something, but then we couldn't turn them into arrays!

Step two: Masking

- We need to find the mean and standard deviation
- We then need to find temperatures that are *outside* of two std. devs.
- Remember, we also need to keep the months for each temperature!
- NumPy has `np.mean` and `np.std`

Step two: Masking

```
mons = np.array(monLst)
temps = np.array(tempLst)
mean = np.mean(temps)
stddev = np.std(temps)
mask = (
    (temps < mean - 2*stddev) |
    (temps > mean + 2*stddev)
)
monsMasked = mons[mask]
tempsMasked = temps[mask]
```

Step two: Masking

```
mons = np.array(monLst)
temps = np.array(tempLst)
mean = np.mean(temps)
stddev = np.std(temps)
mask = (
    (temps < mean - 2*stddev) |
    (temps > mean + 2*stddev)
)
monsMasked = mons[mask]
```

Note that this mask was made from `temps`, but it works fine with `mons` because they're the same size

Step three: Reporting

- Finally, let’s tell the user what months were outliers in a clearer way than just printing the arrays

Step three: Reporting

```
print(
    f"Mean temperature: {mean}. " +
    f"Std. dev.: {stddev}. " +
    f"Expected range: {mean-2*stddev} " +
    f"to {mean+2*stddev}"
)
print("Outlier samples:")
for idx in range(len(tempsMasked)):
    print(
        f"  {monsMasked[idx]}: {tempsMasked[idx]}"
    )
```

Bonus step four: Plotting

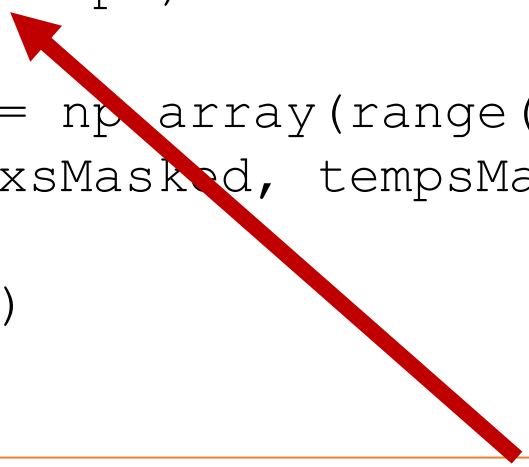
- Let’s plot the temperatures, and highlight in red where they’re outliers
- Plotting the temperatures is relatively easy, but marking the outliers will take some work…

Step four: Plotting

```
plt.plot(temps)

xsMasked = np.array(range(len(temps)))[mask]
plt.plot(xsMasked, tempsMasked, "ro")

plt.show()
```

Step four: Plotting

```
plt.plot(temps)

xsMasked = np.array(range(len(temps)))[mask]
plt.plot(xsMasked, tempsMasked, "ro")

plt.show()
```



Since we didn't provide xs, this just uses 0…len in the x axis

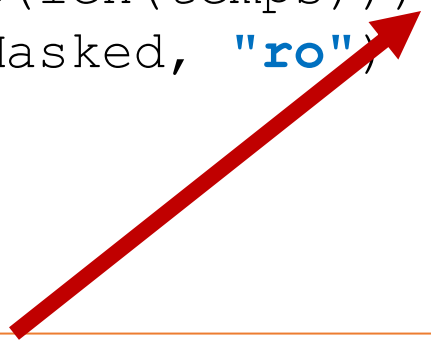
Step four: Plotting

```
plt.plot(temps)

xsMasked = np.array(range(len(temps)))[mask]
plt.plot(xsMasked, tempsMasked, "ro")

plt.show()
```

So, this range is the same xs

Step four: Plotting

```
plt.plot(temps)

xsMasked = np.array(range(len(temps)))[mask]
plt.plot(xsMasked, tempsMasked, "ro")

plt.show()
```



When we mask this array like the temps array, we get the xs associated with every masked temperature

In-lecture quiz (L18)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: If I hadn't masked xs and just used `plt.plot(tempsMasked, "ro")`, what would my plot have shown?

A. Nothing (there would have been an error)
B. The right times would be plotted, but the wrong temperatures
C. The right temperatures would be plotted, but the wrong times
D. The masked xs were not needed; it would plot exactly the same

In-lecture quiz (L18)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What is `a` after this code?

```
a = np.array([-2, -1, 0, 1, 2])
a[a < 0] *= -1
a[a > 0] *= -1
```

A. Nothing (this code has an error)

B. `np.array([-2, -1, 0, -1, -2])`

C. `np.array([2, 1, 0, 1, 2])`

D. `np.array([2, 1, 0, -1, -2])`

E. `np.array([-2, -1, 0, 1, 2])`

Masking and lists

CS114 (M8)

Lists can’t mask

- Remember: masks only work with arrays
- But, we can always convert lists to arrays
- Don’t be afraid to build a mask with append just like any other list!
- Let’s read sparse data (data with missing values) from a CSV and mask the missing values

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/sparse.csv
```

```
import csv
import numpy as np

with open("sparse.csv") as ifh:
    scores = csv.DictReader(ifh)
    fieldnames = list(scores.fieldnames or [])
    for row in scores:
        studentScores = []
        scoreMask = []
        for field in fieldnames:
            val = row[field]
            if field != "Student ID":
                if val == "":
                    studentScores.append(0.0)
                    scoreMask.append(False)
                else:
                    studentScores.append(float(val))
                    scoreMask.append(True)
        studentScoresA = np.array(studentScores)
        scoreMaskA = np.array(scoreMask)
        print(
            "Mean score:",
            np.mean(studentScoresA),
            ", mean of submitted:",
            np.mean(studentScoresA[scoreMaskA])
        )
```

Saving and loading arrays

CS114 (M8)

Administrata note

- This has nothing to do with masking
- It’s here ’cause I had nowhere else to stick it

Saving and loading

- NumPy has its “own” “format” for saving and loading arrays
- Actually, it’s just SSV (space-separated values), but it has convenience functions for saving it!

Saving an SSV

```
a = np.array([1.1, 2.2, 3.3])
np.savetxt("my-vector.ssv", a)
b = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
np.savetxt("my-matrix.ssv", b)
```

Formatting

- 1-dimensional array:
  - One value per line
  - Values written in the most ridiculous format possible
- 2-dimensional array:
  - One row per line
  - Space-separated
  - Values written in the most ridiculous format possible

Loading an ssv

- Don’t forget `csv`! You can do this yourself!
- Or,

```
a = np.loadtxt("my-vector.ssv")
b = np.loadtxt("my-matrix.ssv")
```

Classes

CS114 M9

Administrata note

In *Think Python*, this is Module 8

Putting things in other things

CS114 (M9)

Putting things in other things

- Modules have things in them (e.g., `math.pi`, `np.sum`)
- Values have things in them too (e.g., `str.split`, `arr.shape`)
- This is organizationally helpful, because related things are together
- But it's also helpful for types: you can't accidentally try to split anything but a string, because split is *in* the string!

Putting things in other things

- Putting things in other things isn't just for built-in types: we can make our own types!
- Let's build a type that stores things like a list (in fact, we'll store things *in* a list), but keeps track of the minimum, maximum, sum, and mean as you go

Classes

- Types are defined by classes
- A *class* boxes up functions to make them methods, and values to make them fields
  - Fields are like `arr.shape`: accessible on the values, but not functions, so not methods
- Collectively, methods and fields are the "*attributes*" of the type
- We can make values from the class, and they will have these attributes

Starting our class

```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
```

- Docstrings are like in functions: if I call help(StatsList), I'll get that string
- Fields are just listed with their types

Classes

- When a class is called as a function, it makes a value that has these attributes
  - We usually call such values objects, but…
  - … all the values in Python are defined by classes, so they're all objects! That's not true in other programming languages.
- Let's use our StatsList and see what happens

Where are my fields?

```
x = StatsList()
print(x.lst)  # CRASH! StatsList
              # has no attribute
              # lst!
```

- But I gave it a field `lst`! What's going on?

Initializing classes

CS114 (M9)

Where are my fields?

- Something to remember about type annotations from waaaaaay back: *they're just documentation*
- To have a field there, we have to put it there
- We can add fields just like we add keys to dictionaries. That is, just set them!

There are my fields!

```
x = StatsList()
x.lst = []
print(x.lst)
```

- This works now, but it's a bit unsatisfying
- After all, when we make a NumPy array, we didn't have to make the array, *then* put something in it

The initializer

- Classes can contain methods
- One method is used as the *initializer*
  - (It's called when you create a value in the class)
- The initializer is named `__init__`
  - That's two underscores, then "init", then two underscores